Enabling Interactive Applications over the Internet *

Vijaykumar Krishnaswamy † Ivan Borissov Ganev Jaideep M Dharap

Mustaque Ahamad

College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{kv, ganev, jaideep, mustaq}@cc.gatech.edu

October 20, 1999

Abstract

As computers become pervasive in the home and community and homes become better connected, new applications will be deployed over the Internet. Interactive Distributed Applications involve users in multiple locations, across a wide area network, who interact and cooperate by manipulating shared objects. A timely response to user actions, which can potentially update the state of the objects, is an important requirement. Because of the inherent heterogeneity of the environment, traditionally distributed applications are built using technologies like distributed objects. These technologies are built around a central server paradigm which is undesirable because the response time for the actions of interactive users is always subject to communication latencies. Our approach is to extend these technologies with aggressive caching and replication mechanisms without changing the remote object interface to the applications. Thus, caching and replication are done transparently to provide interactive response time and to improve scalability. A flexible caching framework is presented, where objects can be cached in an application specific manner. It provides multiple consistency protocols that enable tradeoffs between the consistency of a cached object's state at a particular client, and the communication resources available at the client. At runtime, clients can specify their consistency requirements which can vary across different clients. This can be done via a Quality of Service specification interface that is meaningful at the application level. This paper presents the caching framework, its implementation and some preliminary performance results.

Keywords: Remote Method Invocation(RMI), Caching, Consistency Protocols, Timeliness, Quality Objects(QuO).

*This work was supported in part by NSF grants CCR-961937 and a SURA grant.

†Contact author: Email – kv@cc.gatech.edu, Phone – (404) 894-6169, Fax – (404) 385-1253.

1 Introduction

As computers become pervasive in the home and community and homes become better connected, a class of new applications will be deployed over the Internet. We consider applications that we call *interactive distributed applications*. These applications involve users in multiple locations who interact and cooperate with each other by manipulating shared objects. Examples of such applications include collaborative design, distributed games, and educational applications.

The underlying system support that will enable interactive distributed applications must address a number of challenges. First, because the applications are interactive, it is necessary to provide quick response to a user's action even when the users he or she is interacting with are connected by a high latency communication network. Users must also observe remote actions in a timely fashion, and the timeliness requirements could vary across users. This could either be due to differences in available resources at users, or because of the differing roles they play in the application. Finally, because user actions can change the state of shared objects, users must have a consistent view of the objects that are manipulated by them.

Interactive distributed applications can be built using technologies such as distributed objects (e.g, Java RMI, CORBA or DCOM). Although these technologies are attractive for building such applications in heterogeneous environments, they require that shared objects be implemented by common servers, and users must access such an object by remotely invoking it at the server node. Such centralized servers are undesirable because response time for user actions that manipulate the objects cannot be independent of the communication latencies in the system. We are exploring an approach that retains the ease of programming benefits of distributed objects while providing interactive response time to the invocations made to them. This is done by replication and caching of object state where it is accessed. Consistency requirements arise across the multiple copies that are created when replication and caching are employed. We develop a quality of service (QoS) interface that allows applications to specify their consistency needs. For example, a client can specify timeliness requirements to ensure that it learns of a remote update to a cached object within a certain time period after the write is done.

We present a framework for caching Java distributed objects at client sites. The caching framework is integrated in the BBN's quality-of-objects (QuO) architecture that allows QoS needs to be specified in metrics that are meaningful at the application level. The following are the primary contributions of the paper.

1. We develop a framework that allows clients to invoke cached objects transparently. If a user action results in the invocation of one or more objects, often their cached copies can be invoked and hence response time independent of communication latencies, can be provided. The clients only need to specify their consistency needs for the cached object copies, which is done at a high level using QuO's contract object facility.

2. We implement consistency protocols that are particularly well-suited for a heterogeneous environment. In particular, they offer consistency vs. resource usage tradeoffs, and different clients may request different levels of consistency.

3. We develop a prototype system and use it to evaluate the effectiveness of the consistency protocols and characterize the costs of providing a certain level of consistency.

Section 2 describes an interactive distributed application and some interesting properties of such applications. Section 3 presents a brief overview of the system architecture. The consistency protocols and their implementation in the caching framework is described in Section 4. We present performance results in Section 5. Related work is discussed in Section 6 and the paper is concluded in Section 7.

2 Interactive Applications

Interactive applications are those that process user input and respond to user actions on a continuous basis. We consider distributed interactive applications that involve several users in different locations. The actions issued by one user could impact other users and hence their actions. Many such distributed interactive application scenarios can be developed easily. We briefly describe the AquaMoose[6] system that is currently being developed. AquaMoose supports an online community of children interested in educational activities. These children can be geographically distributed and can share and manipulate a virtual world representing an ocean. A user can create various entities in the world and entities created by different users may have rich interactions. For example, two fish can race in trajectories defined by their creators and the bigger fish may eat the smaller one. The virtual world visualization at each user is driven by the state of the entities in the world (e.g., fish) including their location and direction of movement. The entity state changes dynamically as the virtual world evolves.

A closer study of AquaMoose and other such applications reveals some very interesting properties. These applications have state that is highly dynamic. Also, for these applications to perform well, the response time to the user actions should be bounded. For example, a delay of more than 100ms in a direct manipulation interface is perceptible. As the delay for these user actions increases, the user satisfaction with these applications worsens. If the participants for these applications are connected via a wide area heterogenous network with different type of connectivities, the network latencies could be much larger than this threshold. One approach for developing such applications is to maintain a replica of the shared application state on the local machine and keep it consistent with the replicas at other participating sites by using consistency protocols. This way invocations made by user actions can be executed with the local copy. Also, different user actions may require different levels of consistency for their replicas. For instance, in the AquaMoose example, if two different fish are far away from each other and are controlled by two geographically separated users, then the updates made to their attributes (e.g. locations) can be disseminated relatively slowly to the other remote site. But if these fish are in close proximity to each other, then the updates made to their locations should be quickly transferred to the other site for acceptable execution behavior of the application.

In the following sections we will explore the system support for developing such highly interactive and multi-user distributed applications. The operation of the system can be customized based on the application requirements and available resources. In particular, users can specify their consistency requirements easily and differences in user needs can be accommodated. We achieve this goal by devel-

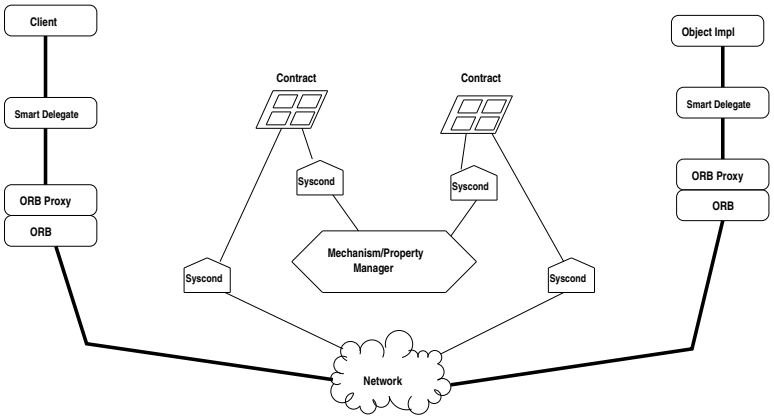


Figure 1: The Quality Object Framework

oping a object caching framework that allows consistency to be specifed as QoS parameters.

3 System Architecture

In a distributed object system, invocation to a remote object requires communication with the server that implements the object. To provide acceptable performance for user actions in interactive applications, it is desirable that the latencies associated with the method invocations be minimized. In wide area systems, a major portion of the invocation time can be attributed to network latencies. This overhead can be avoided by locally caching the state of the objects used for building these applications. Caching can be effective in such applications because of two reasons. First, GUI based visualization of the application is driven by the state of the shared objects. Hence, their state is frequently accessed. Second, updates to cached object state can be disseminated periodically, depending on the consistency requirements of the applications. For example, in the AquaMoose application, the cached state (or computed state based on techniques such as dead reckoning[14]) can be used repeatedly until a new update for the fish's position is received.

We have developed a caching framework for distributed objects, that can transparently cache the objects at the clients that invoke them. The consistency requirements can differ depending on the application needs and where they are deployed. The framework that we have developed addresses this by providing facilities for adding different consistency protocols to the system. This system has been implemented in the *Quality Object*s (QuO) framework. The QuO framework allows for a high level specification of the consistency QoS required by an application. In the following sections we will discuss the QuO's compile time and runtime support for specifying and maintaining the QoS of a cached object state.

3.1 QuO Framework

QuO[29] is a framework that has been developed to support distributed applications with QoS requirements. QuO provides the ability to specify, monitor, and control QoS in an application. In a traditional

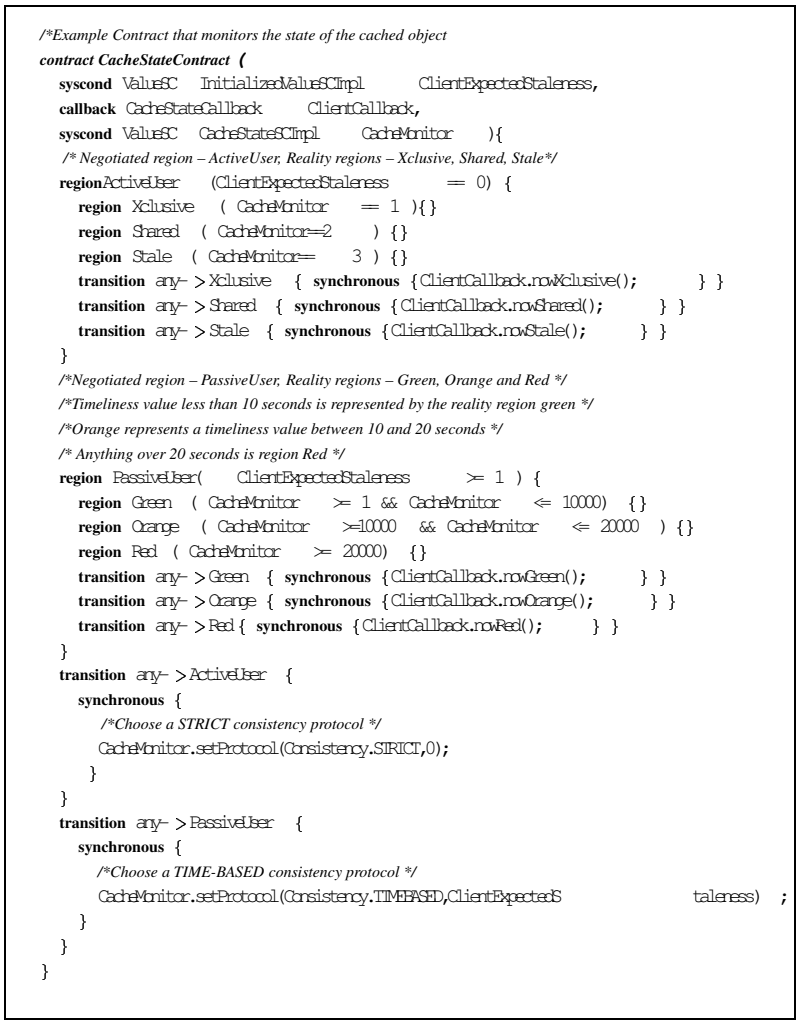
CORBA application, a client makes a method call on a remote object through its functional interface. The call is processed by an object request broker(ORB) on the client's host, delivered to an ORB on the object's host, and executed by the remote object. The client sees it strictly as a functional method call. A QuO application adds additional steps to this process for QoS evaluation as described below. As shown in Figure 1 the QuO application consists of the following additional components.

- A QoS contract between the client and the object. This specifies the level of service desired by the client, the level of service the object expects to provide, operating regions indicating possible measured QoS, and actions to take when the level of QoS changes.
- A local delegate of the remote object. The delegate provides a functional interface identical to the remote object, but can trigger contract evaluation upon each method call and return. The QoS developer can provide alternative behaviors and a dispatch statement which chooses among the alternatives based upon the current state of the contract.
- System condition objects interface between the contract and resources, objects and ORBs in the system. These are used to measure and control QoS.

When a client calls a remote method, the call is passed to the object's local delegate instead. This is transparent to the client, since the remote object and the delegate have the same interface. The delegate can trigger contract evaluation, which accesses the current values of system condition objects measuring different aspects of the system's state. The contract consists of a set of nested regions which describe the possible states of QoS in the system. Each of these regions is defined by a predicate on the values of system condition objects. The contract evaluates the predicates to determine which regions are active and passes the list to the delegate. Contract evaluation can also be triggered by changes in some system condition objects, i.e., those that are observed by the contract. Regardless of how contract evaluation is triggered (by a method call/return or change in a system condition), a transition from one active region to another can trigger transition behavior, which consists of client callbacks or method calls on system condition objects.

A suite of Quality Description Languages (QDL) for describing contracts, system condition objects and the adaptive behavior of objects and delegates is provided by the QuO system. QDL consists of a Contract Description Language (CDL) and a Structure Description Language (SDL). CDL is used to describe the QoS contract between a client and an object, including the QoS that the client desires from the object, the QoS that the object expects to provide, regions of possible levels of QoS, system conditions that need to be monitored, and behavior to invoke when client desires, object expectations, or actual QoS conditions change. SDL describes the internal structure of delegate's implementations, such as implementation alternatives, and the adaptive behavior of object delegates. The object delegate generator creates client-side and server-side object delegate code from SDL, CDL, and IDL code. More details on QuO's architecture can be found in [29].

A simple contract *CacheStateContract*, specifying the QoS regions of operation for an object caching system is shown in Figure 2. This contract is used to continuously monitor the current state of a cached object. CacheStateContract has a few variable declarations. The variable *ClientExpectedStaleness* is



```
/*Example Contract that monitors the state of the cached object
contract CacheStateContract (
  syscond ValueSC InitializedValueSCImpl ClientExpectedStaleness,
  callback CacheStateCallback ClientCallback,
  syscond ValueSC CacheStateSCImpl CacheMonitor ){
  /* Negotiated region – ActiveUser, Reality regions – Xclusive, Shared, Stale*/
  region ActiveUser (ClientExpectedStaleness == 0) {
    region Xclusive ( CacheMonitor == 1 ){}
    region Shared ( CacheMonitor==2 ) {}
    region Stale ( CacheMonitor== 3 ) {}
    transition any->Xclusive { synchronous {ClientCallback.nowXclusive(); } }
    transition any->Shared { synchronous {ClientCallback.nowShared(); } }
    transition any->Stale { synchronous {ClientCallback.nowStale(); } }
  }
  /*Negotiated region – PassiveUser, Reality regions – Green, Orange and Red */
  /*Timeliness value less than 10 seconds is represented by the reality region green */
  /*Orange represents a timeliness value between 10 and 20 seconds */
  /* Anything over 20 seconds is region Red */
  region PassiveUser( ClientExpectedStaleness >= 1 ) {
    region Green ( CacheMonitor >= 1 && CacheMonitor <= 10000) {}
    region Orange ( CacheMonitor >=10000 && CacheMonitor <= 20000 ) {}
    region Red ( CacheMonitor >= 20000) {}
    transition any->Green { synchronous {ClientCallback.nowGreen(); } }
    transition any->Orange { synchronous {ClientCallback.nowOrange(); } }
    transition any->Red { synchronous {ClientCallback.nowRed(); } }
  }
  transition any->ActiveUser {
    synchronous {
      /*Choose a STRICT consistency protocol */
      CacheMonitor.setProtocol(Consistency.STRICT,0);
    }
  }
  transition any->PassiveUser {
    synchronous {
      /*Choose a TIME-BASED consistency protocol */
      CacheMonitor.setProtocol(Consistency.TIMEBASED,ClientExpectedS taleness) ;
    }
  }
}
```

Figure 2: A simple Contract that exports the current state of the cached object to an application defined *Callback* object

used to specify the current timeliness QoS required by the user. *ClientCallback* is a handle to the callback object. This will be invoked whenever there is a discrepancy in requested QoS and the current QoS. *CacheMonitor* object is used to monitor the attributes of the caching framework that are of interest. As seen from Figure 2, the contract is divided into a series of regions. A closer observation will reveal that these are defined at two different levels. The first are the *negotiated* regions, which are the regions of operation that are negotiated for the users. The users specify the region they want to operate in through the variable ClientExpectedStaleness (different users can choose different regions).

In this particular contract example, the negotiated regions are *ActiveUser* and *PassiveUser*. For example in the AquaMoose application described earlier, the entities (fish) may be exclusively owned by users who create them. The attributes of these entities are only changed by the owners, while the other users who are in the vicinity of these entities in the virtual world would like to observe these changes. So the users can be divided into two distinct groups based on their read/write access patterns. One group predominantly reads the shared state of the object while the other one actively modifies it. A user who frequently modifies the state of a shared object can negotiate for the ActiveUser region. PassiveUser region can be negotiated for by a user who mostly reads the state of the shared object. The user can select the negotiated region to be ActiveUser by setting a value of 0 for variable ClientExpectedStaleness in the contract. Any other positive non-zero value sets the negotiated region to be PassiveUser. As seen from the sample code fragment in Figure 2, the specification of a negotiated region triggers a sequence of events. For example, if the user negotiates for the ActiveUser region, then the transition $any -> ActiveUser$ invokes the *setProtocol* method in the *CacheMonitor* object. This sets the consistency policy to the one that guarantees immediate dissemination of the new object values. Transition into PassiveUser region will also trigger events similar to the one described earlier for the ActiveUser region.

The second level of regions are called the *reality* regions. These are the active regions between which the QoS state of the application transitions during program execution. The contract gets evaluated just before and after each method invocation. This evaluation will return the current reality region of operation. In our example, the reality regions are *Xclusive*, *Shared* and *Stale* for the ActiveUser region. They correspond to the state in which the object is currently cached. *Green*, *Orange* and *Red* are the reality regions for the PassiveUser region. They indicate current age (potential staleness) of the cached object. Transition to any reality region triggers handlers in the callback object, which can be used to inform the client application (e.g., via a GUI representation for the state consistency in the callback object).

3.2 Adding Caching to the QuO Framework

In order to cache objects locally at client sites and provide consistency guarantees defined by the contract, the framework should address several important issues. Some of them are the following

- How and when to cache an object. It should be possible to enable and disable caching dynamically based on locality of access and resource availability.

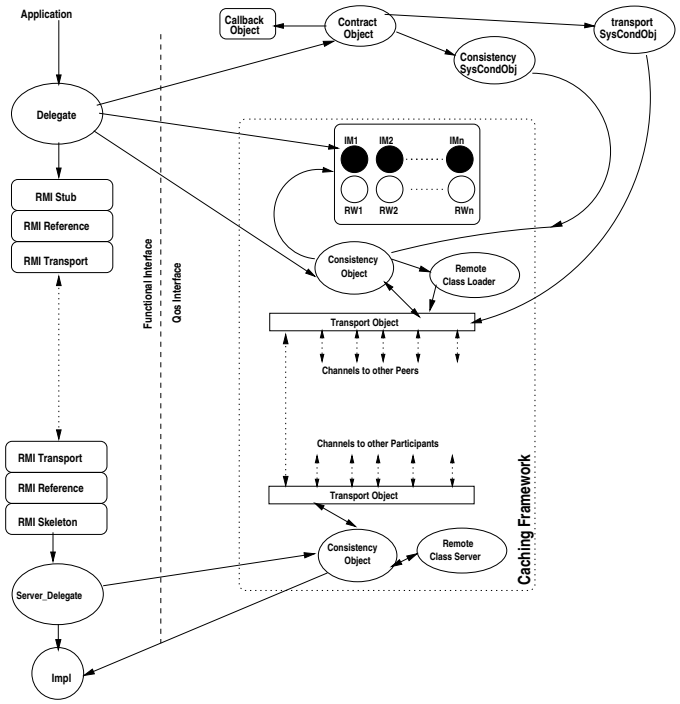


Figure 3: The Caching Framework in the context of the Quality Objects.

- How does the system guarantee the caching QoS requirements of an application?. How are the high level QoS parameters associated with the consistency model parameters to the such as the timeliness threshold, type of ordering etc.

- The consistency actions executed by the protocols depend on the type of access, i.e., if an invocation results in reading of an object's state or is the state also updated. How is read/write access information for the object member functions inferred is an issue that needs to be addressed.

The caching framework that has been developed by us tries to address these issues. Our current prototype has been developed in Java. Some of the objects that make up the caching framework on the server and client sides are shown in Figure 3.

The caching subsystem on the client side consists of a *CConsistencyObject* which is responsible for maintaining the consistency of a cached implementation or implementations. The policy for the CConsistencyObject can be specified through the contract as shown in Figure 2. The caching framework is accessed through a *SmartDelegate* that is specifically created by QuO's stub generators for caching purposes. The smart delegate has two references to the Remote Object. The first one is a direct reference to the cached implementation. This is used whenever the invocation is made on the locally cached implementation. The second one is a *Java RMI* remote object interface to the actual object implementation at the server. This is used to invoke the object at the remote location when caching is disabled.

The objects, RW_1, RW_2, .., RW_n in Figure 3 have the read/write access information for all the member functions of the implementations IM_1, IM_2, ..., IM_n. These objects are generated from the compile

time tags associated with method definitions. The caching subsystem has a *TransportObject* which is used to communicate with other clients and the server. *TransportChannel*s are provided by the TransportObject to communicate with the caching framework created at other clients. The implementation of these channels is optimized for transporting consistency actions.

The server side of the framework consists of a *SConsistencyObject*. This orders the invocations that take place at the server with the ones that are executed with cached copies at client sites. Further, it serves as a rendezvous point for incoming clients, providing them with information to setup their caching framework. If the consistency policy happens to be server based, then SConsistencyObject plays a more active role in keeping the client copies consistent. The *ServerDelegate* redirects all the direct invocations made on the implementation at the server to SConsistencyObject, thus ensuring consistency when both cached and non-cached invocations are executed. *RemoteClassLoader* is used by the framework to remotely load the definitions of the implementations IM_1, IM_2, ..., IM_n, the read-write access information objects RW_1, RW_2, .., RW_n, and other objects that are referred to by the implementations in the server process. *RemoteClassLoader* is the server side component that serves the class definition requests from the RemoteClassLoader.

The following are sequence of actions that are performed when an invocation *m*, is made on the locally cached object.

1. The application invokes the method *m* in the delegate.

2. QuO semantics dictate a pre-method and a post-method evaluation of the contract. The delegate does a pre-method evaluation of the contract object to determine the current QoS region of operation. This can be used to determine the current state of the cached object, staleness value, ownership etc.

3. The contract checks with the system condition objects to determine the status of the cached object monitored and if necessary, it also communicates with the callback object.

4. The delegate consults the consistency object to ensure that the cached object is in a consistent state with respect to the invocation. If the object is not valid or if the invocation mode does not match with the current mode in which the copy is cached, then the delegate asks the consistency object to perform the necessary consistency actions based on its policy to bring the cached implementation to a consistent state. It also temporarily locks the implementation for the duration of the call thus providing method level atomicity.

5. The delegate invokes the method on the local object copy.

6. The delegate then makes a request to the the consistency object to free up the resources allocated for the call. The consistency object unlocks the object thereby allowing any pending consistency actions to be executed.

7. During the post-method evaluation, the delegate once again communicates with the contract to determine the active region.

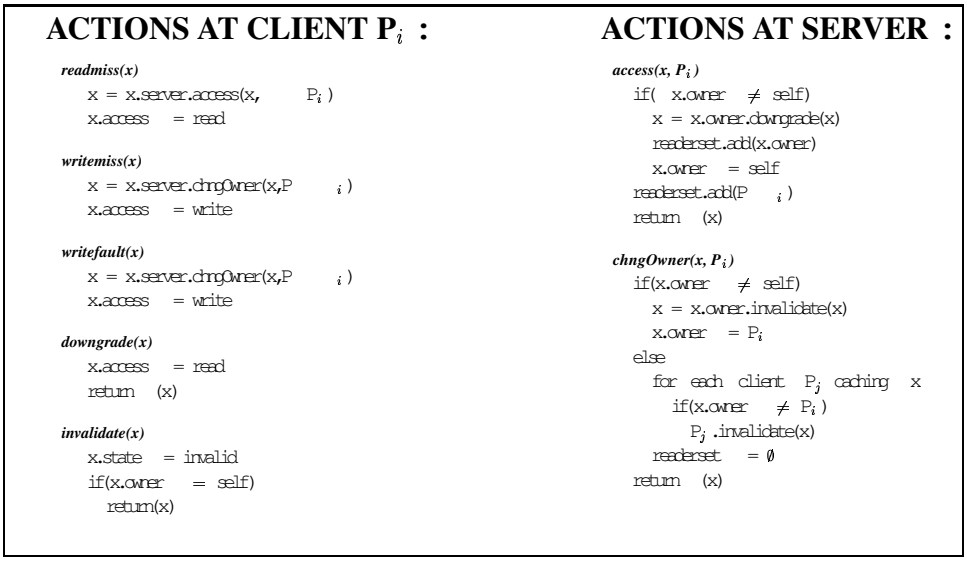


Figure 4: Central Server Invalidation protocol. This is the base protocol.

8. The contract probes the system condition objects for new values.This may result in signalling to callback objects. This can be useful for post method operations like starting new consistency actions in the background without blocking the application.

9. The delegate finally returns the results to the application.

4 Consistency Protocols

Consistency protocols ensure that client invocations are executed with local object copies that have consistent state. The consistency of a cached copy of an object can be defined along two dimensions, namely *orderliness* and *timeliness*. The orderliness property specifies how updates to the object are ordered and viewed by read operations. For the convergence of an object's state (e.g., a unique final state of the object is obtained after the execution of the invocations), it is essential that all writes to it are ordered. Weaker orderings are possible(e.g. causal[2]) when concurrent writes are rare or other mechanisms can be used to fix the divergent object state. The timeliness property specifies the time interval after which an update to an object must become visible at sites that are caching it. By increasing the timeliness interval, updates can be propagated to remote sites with decreased frequency which reduces the number of consistency messages. These two dimensions of consistency are independent. For example sequential consistency provides strong ordering but no timeliness guarantees.

In this section we describe some of the consistency protocols that we have implemented in the caching framework to maintain the consistency of the cached object state. The first one is an invalidation based protocol (SC_{inv}) and the second one, LC_{set}, is the local consistency (LC) protocol which is based on invalidation sets. The two protocols present different approaches to maintaining cache consistency, and allow us to quantify the reduction in overhead when timeliness is varied to save communication resources. SC_{inv} is an example of a strong consistency protocol that provides a unique ordering for all

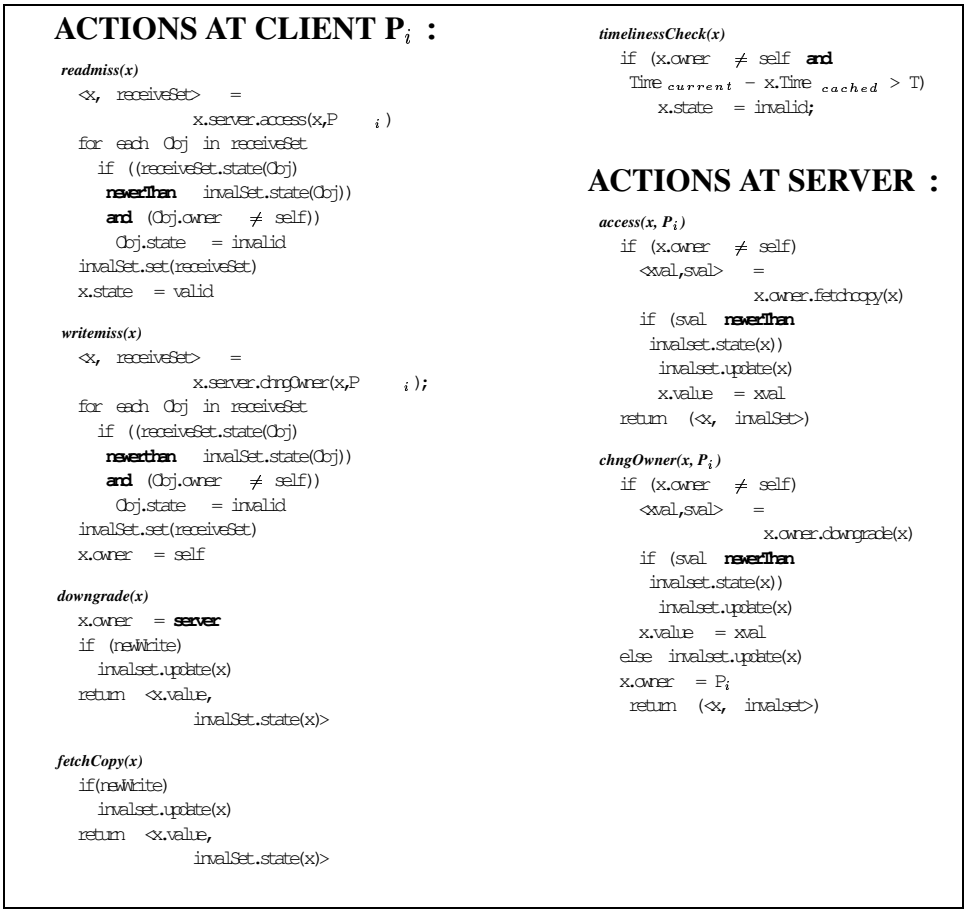


Figure 5: Invalidation-Set protocol.

the writes in the system while providing immediate timeliness for the reads. It is used widely in shared memory systems [8, 19] as well as file systems [26, 23, 9]. LC_{set} allows the timeliness threshold to be varied, but orders writes as in SC_{inv}. By changing the timeliness threshold for reads, LC_{set} allows the consistency overhead of cached objects to be varied according to the needs of the application sharing of the objects.

Because our framework works at the object level, for the following discussion we consider method invocations on cached objects as writes or reads, depending on whether the method modifies the object or not. We now proceed to describe the details of the SC_{inv} and the LC_{set} protocols.

4.1 Server based Invalidation protocol (SC_{inv})

A server based invalidation protocol, at a given time allows either a single writer or multiple readers. We refer to the client that caches an object copy in exclusive mode as its *owner*.

When a client P_i attempts to read an object copy and experiences a read-miss, it communicates with the server. If no other client caches the object copy in exclusive mode, then the server returns its copy to the client and adds the client to its reader set. Otherwise, the server downgrades the owner's copy to read-only mode and provides P_i the latest copy from the owner. For a client P_i to perform a write operation on an object *x*, it needs to cache it in an exclusive mode. If *x*'s copy has not already been cached, P_i experiences a write-miss. On the other hand, if a copy has been cached in a shared mode, P_i experiences a write-fault. In either case, P_i communicates with the server. In case of a write-miss, the server returns its copy of *x*, immediately, if the object is currently not cached by the other nodes. If copies exist at other nodes in shared mode, then the server invalidates all such copies, and returns *x* to client P_i in an exclusive mode. If another node happens to cache *x* in an exclusive mode, then the server gets the recent state from that node, invalidates that copy and finally sends the most recent copy to the requester. Although this protocol orders all writes and provides immediate timeliness, its scalability, however, is limited because of the high communication costs of synchronous invalidations for update operations. The protocol presented next attempts to alleviate some of these problems.

4.2 Invalidation-Set protocol (LC_{set})

The LC_{set} protocol was first presented in [1] but we have improved it in a number of ways. Similar to SC_{inv}, LC based protocols also assume a single writer for an object at a time, but there are some important differences between the invalidation protocol and those based on LC.

The LC_{set} protocol permits control over the timeliness aspect of the consistency of cached objects. The protocol strictly orders all the writes that are executed on a group of related objects. However, it allows a single writer to update the object state while other clients are accessing the older state of the object in read only mode. The writes are not immediately propagated to all the remote sites that are caching a copy of the object. This delay may result in the reads at the remote sites returning stale values of the object state. The protocol guarantees that the reads will never return a value for the object that is older than the timeliness threshold specified for the protocol. The clients accesses that return older cached copies are serialized before the writes that create the new value. Below we list some of the

major features of the local consistency protocol, and also highlight how it differs from the invalidation protocol SC_{inv}.

1. In LC based protocols, when a server transfers write access to a new client node, existing readers are not sent invalidation messages. In fact, readers of an object can coexist with a writer by reading values from old cached copies. Thus, not only invalidation messages are avoided, but a node does not need to receive and process messages each time an object cached by it is updated at another node.

2. LC based protocols order all access to related objects by introducing new object copies into the node's cache in a consistent manner. At the time a new object copy is added to a node cache, the node performs local consistency actions to ensure that currently stored copies of shared objects are valid with respect to the information received with the newly fetched object. Such consistency actions utilize meta-data received from the server and require no communication. They only invalidate cached copies that have been potentially updated after they were acquired by the node. The meta-data received by a client includes information about those updates that are known to the server and which were not sent to the client.

3. A client node must communicate with an object's server when it either does not have the object in its cache or it wants to update the object but has only read permission. When such communication takes place, the server can make the client aware of the new updates to objects that are cached by the client. In addition, a client can choose to periodically communicate with the server to learn about new updates. This periodic communication ensures that the timeliness of client copies is always within the specified bounds. Thus, while guaranteeing that object access can be ordered, LC based protocols offer the flexibility to control the timeliness of cached object copies based on resource availability and application needs.

The LC_{set} protocol maintains consistency of cached copies by receiving information about the objects stored at the node that have been overwritten since the node's last communication with the server. The server maintains such meta-data about updates to objects and sends it to a client whenever the client communicates with it. There are two cases in which it becomes necessary for the client to communicate with the server. First, all misses/faults require communication with the server. The server does not send messages to invalidate other copies in the system when a write-miss or a write-fault request is received. Instead, information about the object that needs to be invalidated is recorded in an *invalSet* (for invalidation set).

As shown in the *chngOwner* and the *access* methods in Figure 5, if a client node's (say P_i's) request for an object *x* results in the server contacting the owner of the object, and if the server determines that the recent copy of *x* from the owner is newer than its copy, then this information is used to update the *invalSet*. When P_i's request returns, *invalSet* is piggybacked with the result of the request. The second case when a client communicates with the server is when the client timeliness threshold for an object expires. In this case the client invalidates the cached object copy, but it does not immediately

communicate with the server to request the new copy of the object. A subsequent invocation on this object triggers communication with the server.

Read misses are handled as in SC_{inv} except that the owner node's copy is not downgraded. Instead the server gets the latest copy of the object from it and allows it to continue as the owner. Also, the invalset is returned to the client as in the case of write misses/faults. When a client receives an invalset from the server, it invalidates the cached copies of objects listed in the invalset that have newer state compared to the currently cached ones. Thus, object copies at a client are invalidated only when the client communicates with the server and not when the object is written.

The function *timelinessCheck* from the client side ensures an upper bound on the staleness of the cached copies of the objects at a node. The timeliness requirements are specified on a per-object basis and should be chosen by the application depending on its needs and bandwidth availability. The client nodes keep track of the time elapsed since its last communication with the server for each object, and if that exceeds the timeliness threshold the application is willing to tolerate, then the cached copy is marked invalid. Any future reference to the object will result in communication with the server, which, either provides the latest copy of the object in the system or the cached copy is made valid again.

4.3 Implementation of Consistency Protocols

In this section we describe some of the interesting issues that we addressed in the implementation of all the consistency protocols developed for the caching framework. Details of the caching framework are discussed first and the implementation issues specific to a particular protocol are presented in more detail in the subsequent sections.

Consistency protocols implemented by the consistency objects on both the client and server sides extend the *ConsistenctModel* class and implement the *ConsistenctyScheme* interface. The *Consistency-Model* Class has a generic set of routines that are suitable for any consistency protocol. The *ConsistencyScheme* interface defines methods, through which the consistency object is accessed. In addition, the client side consistency object implements the *ConsistencyScheme_Client* interface and the server side object implements the *ConsistencyScheme_Server* interface. These interfaces contain methods specific to the client side and the server side of the caching framework.

The *ConsistencyModel* is an abstract class. It contains definitions for the methods that are common to all the consistency protocols and abstract methods which are more protocol specific. Some of these methods are the following.

- *setProtocol* - It is used to set the consistency policy of the caching framework.
- *fetchLatestCopy* - It ensures that the reference to the cached copy is in a valid state. It is an abstract method, whose definition is delegated to a particular consistency protocol object.
- *takeActions* - It is invoked when a request for a consistency action is received from other consistency objects at server or at other clients. The actual definition of the method is delegated to the protocol object.

The *ConsistencyScheme* interface provides access points through which the caching framework can be accessed from the outside (e.g. delegate). It declares the following two functions:

- *guard*- It is invoked before the actual method invocation. This method locks the object so that no consistency related actions are performed on the object during the execution of the method.
- *relax* - It is invoked after the execution of the method is completed. It releases the lock acquired during the guard method .

The *ConsistencyScheme_Client* interface declares the method cacheObject, which is called once when the object is cached for the first time at the client node. The method adds the object to the set of objects cached at the client. The *ConsistencyScheme_Server* interface declares the cacheObject method which is called once when an object is exported at the server node.

4.3.1 Server based Invalidation protocol

This section describes some of the data structures and implementation details that are specific to the server based invalidation protocol. The client side protocol object extends the ConsistencyModel class and implements the ConsistencyScheme and the ConsistencyScheme_Client interface. The corresponding server side object extends the ConsistencyModel class and implements the ConsistencyScheme and the ConsistencyScheme_Server interface. It allows multiple readers or a single writer for a given object. The server maintains information about the identity of readers or the writer which is used for consistency actions. The *reader-set* data structure is used to store information regarding the readers. It is maintained as a vector. The readers can be added, removed and enumerated from the reader-set. It grows and shrinks based on the accesses at the server and the clients. If there is a sequence of read-misses from different clients, then the server adds the reader identities to the reader-set. Whenever there is a write-miss or a write-fault, all the members of the reader-set are asked to invalidate their local copies and the reader-set becomes empty.

4.3.2 Invalidation-Set protocol

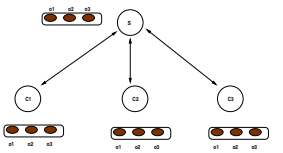
This section describes some of the implementation details that are specific to the invalidation-set protocol. This protocol allows a single writer and multiple readers of a shared object to co-exist. Hence an object may get modified, even when it is available for reading at other clients. Since a client does not get notified when objects cached by it are updated, the server side consistency object maintains information about updates which are known to the server but not propagated to the client. In particular the server stores the identites of the objects that have been updated since a client last communicated with the server. The data structure that stores such information is called an invalidation-set. To maintain consistency, the invalidation-set is sent to the client whenever it communicates with the server. On receiving the invalidation-set, the client invalidates the objects listed in the set and requests their new copies from the server when it faults on them during future invocations. The client communicates with the server and receives a invalidation-set only when it experiences a read-miss or a write-miss. If these misses/faults do not occur for a while, then the timeliness guarantees requested by the client may not

be met. Thus, there has to be some mechanism to ensure that the client receives the invalidation-set before the timeliness period expires. Also, the framework at any time has multiple threads trying to access the cached object and other meta-data associated with the cached object. These threads have to be synchronized in order to consistently access the object state. The following data structures were devised to address these issues.

There are two different ways in which the server can maintain this invalidation-set. It can either maintain the set on a per-client basis or can have a single invalidation-set for all the clients. We choose the second approach. The reason for this choice is that it would scale better if the system had a large number of clients. This is because of the reduction in the computation costs associated with updating each client's invalidation-set, when there are a large number of them.

If a single invalidation-set is maintained for all clients, we must address the problem of some clients having older object copies while the others having the current copy. The invalidation-set can be viewed as a table of records indexed by the object-id. Each record in the table is the tuple $\{$ *modify-bit, epoch*$\}$. A value of 1 for the modify-bit indicates that a object is currently cached in an exclusive mode by a client. If the invalidation-set does not contain an entry for an object, then it means that either the object has not been cached by any client or is being cached in read-only mode by all the clients. The epoch number can be considered as a generation id indicating how old the object is. The server increments this epoch number whenever it receives a new copy of the object from its current owner. The clients locally maintain an epoch vector for all the objects they cache. Whenever a client receives the invalidation set, it compares the epoch received for each object with its local epoch number and it invalidates the cached copy only if the new epoch number is greater than the local one. This ensures that the client does not invalidate an object if it has its latest copy .

An example of how the invalidation-set is used by LC$_{set}$ to ensure ordering is shown in figure 6. There are three clients C_1, C_2 and C_3 and the server S. The application has three shared objects, O_1, O_2 and O_3 that get instantiated at the server at the start. The figure also shows the sequence of operations executed at the clients. The table in figure 6 gives a possible global serialization order of the operations as provided by the LC$_{set}$ protocol, the consistency action description, the state of the cached objects at the clients before and after the execution of consistency actions, and the value of the epoch numbers in the invalidation-set at the clients after the completion of consistency actions. The clients first populate their caches with the initial state of O_1, O_2 and O_3 which are respectively v_1, u_1, and z_1. As seen from the figure, the operation c_1*:w(*O_1*)*v_2 at client C_1 triggers the consistency action $C_1 \rightarrow S$, which grants the exclusive ownership of O_1 to C_1 . It also changes the invalidation-set at the C_1 and Server to *100*, indicating that the cached O_1 has been modified to a new value. During the write operation c_2*:w(*O_2*)*u_2 at C_2, the consistency operation $C_2 \rightarrow S$, modifies the invalidation-set at C_2 to [*110*]. Since the epoch of O_1 in the received invalidation-set is higher than the locally stored epoch, C_2 invalidates its local copy of O_1. During the next operation c_2*:r(*O_1*)*v_2, C_2 experiences a read miss. The consistency action $C_2 \rightarrow S \rightarrow C_1$ brings the new state of O_1 to C_2 from C_1. Also during this transfer, a new invalidation set *110* is propagated to C_1. Again C_1 invalidates O_2 because of the higher epoch number received. When C_3 writefaults during the operation c_3*:2(*O_3*)*w_2, S sends the invalidation-set [*110*] to C_3. C_3 invalidates the local copies of O_1 and O_2 because their new epoch numbers are greater than the local epoch values.



c1:r(01)v1, r(02)u1, r(03)z1, w(01)v2, r(02)u1, r(03)z1
c2:r(01)v1, r(02)u1, r(03)z1, w(02)u2, r(01)v2, r(03)z1
c3:r(01)v1, r(02)u1, r(03)z1, w(03)z2, r(01)u2, r(02)u2

Global Order provided by LC_{set}	State Before Action				Consistency Actions	New invalidation-set at				State After Action			
	At	O1	O2	O3		C1	C2	C3	S	At	O1	O2	O3
c1:r(01)v1	C1	U	U	U	C1->S	o o o	null	null	o o o	C1	S	U	U
c1:r(02)u1	C1	S	U	U	C1->S	o o o	null	null	o o o	C1	S	S	U
c1:r(03)z1	C1	S	S	U	C1->S	o o o	null	null	o o o	C1	S	S	S
c2:r(01)v1	C2	U	U	U	C2->S	o o o	o o o	null	o o o	C2	S	U	U
c2:r(02)u1	C2	S	U	U	C2->S	o o o	o o o	null	o o o	C2	S	S	U
c2:r(03)z1	C2	S	S	U	C2->S	o o o	o o o	null	o o o	C2	S	S	S
c3:r(01)v1	C3	U	U	U	C3->S	o o o	o o o	o o o	o o o	C3	S	U	U
c3:r(02)u1	C3	S	U	U	C3->S	o o o	o o o	o o o	o o o	C3	S	S	U
c3:r(03)z1	C3	S	S	U	C3->S	o o o	o o o	o o o	o o o	C3	S	S	S
c1:w(01)v2	C1	S	S	S	C1->S	1 o o	o o o	o o o	1 o o	C1	X	S	S
c1:r(02)u1	C1	X	S	S	None	1 o o	o o o	o o o	1 o o	C1	X	S	S
c1:r(03)z1	C1	X	S	S	None	1 o o	o o o	o o o	1 o o	C1	X	S	S
c2:w(02)u2	C2	S	S	S	C2->S	1 o o	1 1 o	o o o	1 1 o	C2	I	X	S
c2:r(01)v2	C2	I	X	S	C2->S->C1	1 1 o	1 1 o	o o o	1 1 o	C2	S	X	S
c2:r(03)z1	C2	S	X	S	None	1 1 o	1 1 o	o o o	1 1 o	C2	S	X	S
c3:w(03)z2	C3	S	S	S	C3->S	1 1 o	1 1 o	1 1 1	1 1 1	C3	I	I	X
c3:r(01)v2	C3	I	I	X	C3->S->C1	1 1 1	1 1 o	1 1 1	1 1 1	C3	S	I	X
c3:r(02)u2	C3	S	I	X	C3->S->C2	1 1 1	1 1 1	1 1 1	1 1 1	C3	S	S	X

U - Uncached X - Exclusive S - Shared I - Invalid

Figure 6: An example of Invalidation-set protocol.The figure shows the new invalidation-sets at the three clients C_1, C_2 and C_3 and the server$S$ when the shared objects O_1, O_2 and O_3 are read or written. The table shows the global order provided by the LC$_{set}$ protocol, the consistency actions needed when operations are executed, the state of the cached objects before and after consistency actions and the invalidation-set epoch values at the clients after the completion of the consistency actions.

Subsequent read operations on them experience read misses and the new states of O_1 and O_2 are brought in from C_1 and C_2.

Clients that do not get notification about the update will continue to read the old copy until they communicate with the server. These reads are ordered before the update in the serialization provided by the LC$_{set}$ protocol. To guarantees that the cached copy of an object does not remain stale for more than the specified timeliness threshold, periodic communication between the client and the server is necessary. This could be ensured in the following two ways.

- The server could maintain a global time clock and after every refresh-period force the clients to invalidate the locally cached copies, by sending them the latest invalidation-set. This is a push based approach.

- In a pull based approach, each client can locally timeout when the communication has not taken place with the server for a certain period of time. The client can then locally invalidate the objects with expired timeliness so that the next invocation on them will result in the client communicating with the server and getting a more recent invalidation-set.

We chose the second option to provide the client, control on when to fetch the object. The client can delay the fetching till the new state of the object is absolutely necessary. Also, this way different clients can choose different timeliness thresholds for their cached copies. In our implementation, whenever an object state is renewed at the client, it also gets timestamped. During every subsequent invocation, the timestamp is compared with the current time. If the current time happens to exceed the timestamp by

more than the current timeliness threshold, then the client is forced to communicate with the server to retrieve the latest copy and in the process also receives the invalidation-set for the entire set of cached objects.

A *Mutex-lock* data structure is used for synchronizing access to cached object copies. At any instant either a application thread or the consistency thread is allowed to access the object. This synchronization among the threads ensures that the consistency actions will not update the object state in the midst of an invocation and vice versa. Thus, consistency actions are delayed if an invocation is currently in progress and similarly the invocation must wait till the processing of a consistency action completes. Mutex-lock can be acquired and owned by only one active thread at any instant. It is released after the operation completes . During the method invocation sequence as described in the earlier sections, the methods guard and relax are called by the application thread. A mutex-lock is acquired during the invocation of the guard method and is releasd when the relax method is invoked. Similarly, the *takeactions* method called by the consistency thread acquires the same mutex-lock before it performs any consistency actions on the cached objects.

5 Performance

So far we have discussed the architecture of the caching system and the consistency protocols that have been implemented to ensure that operations are executed with consistent object state. The goal of this section is to experimentally evaluate the performance of the caching system to characterize the improvements that can be achieved by caching, and the impact of the consistency protocols on the performance of caching. We first measure the costs associated with the basic operations in the system such as the overhead imposed by the caching framework to perform a read and write operations This is followed by a more detailed evaluation of the system with a synthetic workload that was developed based on attributes of an interactive distributed application.

The experiments were conducted on a cluster of 248 MHz Sparc Ultra-30's connected by a 100 Mb Ethernet. The machines were all equipped with 128 Mb of memory. The Java virtual machine used was Java2 from Javasoft and we used it with the just-in-time (JIT) option enabled. There were no other applications running on the machines when the experiments were conducted and hence the numbers generated were repetitive. We ran each of the experiments three times and the numbers presented here are averaged across multiple runs and over multiple clients. It was difficult to generate numbers that were repetitive for a wide-area configuration. This was primarily due to our lack of control on the network. Because of this reason, we are only presenting the measurements for the local-area environment in this paper. Since the benefits of caching will be more pronounced when communication latencies are higher, the wide-area performance should be even better.

In our experiments clients invoked the object *O* implemented by the server. The definition of O has two member functions: *read()*, and the *write()* method. The read method has a null body while the write method increments the state of a shared counter. Since there is a very small amount of time spent in the execution of the methods, the average invocation time obtained in the experiments is a direct measure of the overheads due to communication and computation costs associated with the caching framework.

Invocation Execution	Invocation Time in milliseconds
At remote server	1.244
At locally cached copy	0.025

Table 1: Comparison of the time per invocation in milliseconds averaged over 10,000 invocations. The invocations were executed at the server and with locally cached copies. The size of the object was 1024 bytes and a group of 8 related objects were used.

We first discuss the basic performance of the protocols and then follow it by a discussion of the system's performance for the synthetic workloads.

5.1 Basic Performance of Caching

Object caching enables a remote invocation to be completed locally when the cached object is in a valid state and with proper ownership. If the cached copy is not consistent, the invocation will result in the client communicating with the server to fetch the current copy which could generate one or more messages between the server and clients. To measure the cost of completing a remote invocation in different situations, we decided to measure the cost of invocations in the following cases.

- The object is invoked remotely at the server without caching it locally. Thus, this will be the base case where the existing Java RMI framework is used to make the invocation at the remote server.
- The object is cached locally and is in a valid state and correct mode. This invocation is local and will not result in any communication with the server. This is the best case for caching.
- The object is locally cached but is not in a valid state, but a valid copy can be found at the server. This will result in client communicating with the server, fetching the new copy from it and then executing the invocation locally.
- The object is locally cached and is not in a valid state. A valid copy can be found at some other remote client C_n. This will result in the client communicating with the server followed by the server communicating with the C_n, which will return a copy via the server to the requested client.
- The object cached locally in shared mode and the invocation needs an exclusive copy. The client has to communicate with the server, and the server with other remote clients depending on the protocol.

Table 1 compares the costs of executing an invocation at the server and on the locally cached copy. Eight objects, each of size 512 bytes were used in the evaluation. The average cost of a local invocation is about 25 microseconds, while it takes 1.244 milliseconds for the remote execution to take place at the server. Thus there is a fifty fold improvement in performance if the execution can be done locally. Table 2 compares the costs of similar actions, for cached invocations when SC_{inv} and the LC_{set} protocols are employed for consistency maintainence. For a reader to fetch a new copy from the writer through

Consistency actions for Invocation Execution	Protocol	Size of Reader-set			
		0 Client	2 Clients	4 Clients	8 Clients
With write-miss and copy fetched from server	SC_{inv}	3.286	5.631	8.124	13.532
	LC_{set}	3.575	3.575	3.575	3.575

Table 2: Comparison of the time per invocation in milliseconds averaged over 10,000 invocations. The invocations were executed at the cached object with a write-miss resulting in the new state being fetched from the server. The Size of the Object was 1024 bytes and the group had 8 objects. The *Reader-set* size corresponds to the number of other clients in the system that have the object cached in the shared state.

Consistency actions for Invocation Execution	Object Size in bytes	Invalidation-Set Size				
		1 Object	2 Objects	4 Objects	8 Objects	16 Objects
With read-miss and copy fetched from server	8	2.902	2.961	2.900	3.112	3.525
	1024	3.296	3.309	3.401	3.575	3.955
With read-miss and copy fetched from remote client via the server	8	5.477	5.482	5.488	5.830	6.421
	1024	5.859	5.952	5.959	6.328	6.912

Table 3: Time per invocation in milliseconds averaged over 10,000 invocations for the LC_{set} protocol

the server, it costs about 3.575 milliseconds when using the LC_{set} protocol. Thus the caching system would perform better if we could achieve a hit-ratio greater than or equal to 75%

Table 2 also shows an interesting difference in the two consistency protocols. As the size of the reader-set increases, there is an increase in the invocation time for SC_{inv}, while it does not change for the LC_{set} protocol. This is because SC_{inv} allows either one exclusive owner or multiple shared readers to co-exist. Therefore, before it can grant access to an exclusive copy for a write-miss, it has to invalidate all the clients of the reader-set. But the LC_{set} allows one exclusive writer and multiple shared readers to co-exist at the same time. A request for write-miss does not result in any immediate invalidation messages from the server to the reader clients. Hence the invocation cost does not vary with the number of clients caching the object in the shared mode.

Table 3 compares the invocation cost for the LC_{set} protocol for objects of different sizes (8 bytes and 1024 bytes). There is a 20% increase in the invocation time when the size of the invalidation-set is increased from 1 to 16 objects. This can be attributed to the additional computation and communication costs involved in marshalling and unmarshalling larger invalidation-set objects.

5.2 Workloads

5.2.1 Workload Modeling

The basic performace evaluation clearly reveals that significant improvements can be achieved if the invocations are executed with cached copies. However, locality of access, which determines when

the cached copies can be used, depends on the behavior of the applications. Hence it is desirable to evaluate the system using actual distributed applications. Unfortunately currently available traces are mostly from the distributed file system [5, 9, 23, 26, 10] and the world-wide web [24, 25] domains. The read/write sharing patterns for these are more coarse grained and often there is a single writer for a given object. Hence we chose not to use these traces to evaluate our system. Since interactive applications can involve actual users and their behavior can depend on response time for their actions, it is difficult to create real traces [4]. We decided to use synthetically generated workloads based on important parameters of interactive applications like Aquamoose described earlier. Our synthetic workload can be described by the following parameters. The values associated with the parameters below were those used in generating the traces.

Number of Objects: There are *N* shared objects O_1, O_2, ..., O_n each of size S_1, S_2,..., S_n bytes that are governed by the consistency protocol. They are all instantiated at the same server. In our experiment we assigned a value of 32 to N and all of S_1, S_2,..., S_n were assigned a value of 64.

Number of Clients: There are *C* clients that can make invocations on the objects. We assigned a value of 8 to C.

Number of Invocations per Client: Each client makes *K* invocations. K was choosen to be 10,000.

Read Frequency: Assuming that interactive applications are visual and require frequent screen updates, we generated read request to the objects once in every 30 milliseconds. This roughtly corresponds to a refresh rate of 30 frames per second.

Write Frequency: The writes in these applications may be because of user actions or because of movement of autonomous entities (e.g., movement of fish in a predetermined trajectory). We also assumed that a user does not recognize and differentiate events happening in a time period less than 100ms. So the lower limit for the writes is 100ms (for autonomous entity movement) and the higher limit was fixed at 3 seconds (for user actions). The writes were generated at random with the above mentioned higher and lower time bounds.

Ownership: The ownership was assumed to be static for this trace. This is a reasonable assumption because in many distributed applications like the virtual world, the changes to object state are made only by the users who created them.

We use the same set of traces generated to evaluate the two protocols.

5.2.2 Performance Evaluation

The performance metrics that are of interest are the average invocation time and the number of cache misses and server requests. The experiments were conducted in the same lab environment in which the basic benchmark tests were done. The system was evaluated with 32 objects each of size 64 bytes, all instantiated at the same remote server. There were 8 clients that generated read and write invocations on these objects.

Figure 7 shows the average execution times for invocations at the server, locally cached invocation and local invocation with the LC_{set} protocol, for different values of the timeliness threshold. The average invocation time for execution at the server is 12.52 milliseconds. This is different from the micro-

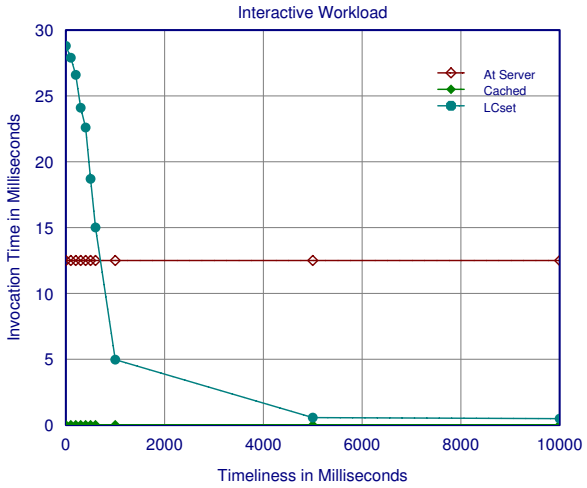


Figure 7: The invocation time averaged over 10000 invocations with 8 clients and 32 objects. The invocation time for the method execution at the server is 12.52 milliseconds while the invocation time for the method execution on the local copy is 67.8 microseconds. LC_{set} invocation time exponentially decreases as the timeliness threshold is increased.

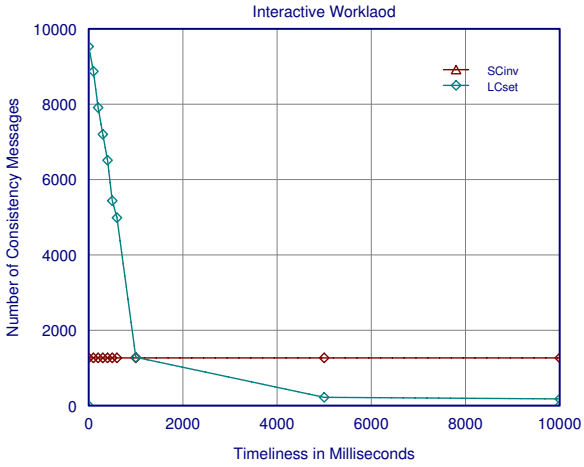


Figure 8: The number of cache misses for 10000 invocations with 8 clients and 32 objects. The cache misses generated for the SC_{inv} protocol is 1267. LC_{set} cache misses exponentially decreases as the timeliness threshold is increased.

benchmark results and can be attributed to the increase in the server load because of 8 clients operating simultaneously. While the average invocation time for a locally cached copy is only 25 microseconds, the invocation time for the LC_{set} protocol is an exponentially decreasing curve. It is about 28.84 milliseconds when the timeliness threshold is 0 and reduces to about 746 microseconds as the timeliness threshold value is increased to 5 seconds. This can explained from the way the protocol works. The clients invalidate their local copies whenever they receive an invalidation-set from the server or whenever the timeliness threshold expires. When the timeliness threshold is set to 0, the object copy expires as soon as it is locally cached. Hence all the invocations find the copy invalid, leading to consistency actions. This accounts for higher method execution times. However, when the timeliness threshold is increased, such invalidation and the resulting communication is decreased significantly. The system initially invalidates the objects when it receives the invalidation-sets. As the steady state is reached, the cache gets populated with all the objects in valid state. So invalidation frequency due to the receipt of invalidation-set decreases and beyond a point, the invalidations are only due to the expiration of the timeliness threshold. This explains why the number of cache-misses decrease as the timeliness threshold is increased, leading to very small execution times for the invocations.

Figure 8 shows the number of cache misses for the protocols SC_{inv} and LC_{set}. The number of consistency messages generated by the SC_{inv} protocol does not depend on the timeliness value. While for the LC_{set} protocol, it is an exponentially decreasing curve. At a timeliness threshold of 0, every invocation other than the writes (because of the exclusive ownership assumption) will result in a cache-miss for reasons explained earlier. This is why we experience 9532 cache misses for the 10000 invocations exe-

cuted during the experiment. As timeliness threshold is increased, a lot more invocations are completed with the cached object and hence the cache misses decrease.

6 Related Work

Distributed Visualization applications like *Spline*[3], *Aquamoose* [6] and other virtual reality (VR) applications have been devloped in the recent past which involve large numbers of geographically distant users interacting in real time. In addition to interacting with each other, users also interact with computer simulations which range from the very simple (e.g., a revolving door) to the very complex (e.g., a human-like robot). These applications also allow users to make temporary and permanent modifications and extensions to the environment while they are running, so that the content of an environment can dynamically grow. For reasons of efficieny, consistency, scalability, these applications are built on abstractions provided by a distributed communication infrastructure. Some examples are of such infrastructures and their features are as follows: *MR* toolkit [27] and *dVS* system [13] assume a client-server topology, where a client is represented by a collection of processes. It uses message passing paradigm to disseminate information between the participants. Any change in the state at a client is propagated by the clients *network* process to every other client's network process. *DIVE* [16] uses shared memory paradigm and uses object abstractions for the shared state. Ownership protocol and multicast mechanisms are used to maintain the consistency of the state. *SIMNET* [7] and *NPSNET* [22] are used to develop military battlefield simulations. They use a combination of shared memory and message passing mechanisms. They use best-effort broadcast or multicast to communicate user actions to remote sites. Scalability is achieved by locally maintaining a copy of remote state and simulating remote actions through one of the allowed set of behaviors of the application. *High Level Architecture (HLA)* [11] is a system where the messages across the nodes can be ordered at the receiver using one of the following ordering types: receive order, priority order, causal order, time stamp order etc. Our distributed object caching system also provides a distributed platform that can be used by the applications mentioned earlier. But it differs from them by allowing clients to specify their timeliness constraints through a high level QoS interface. By separating out the QoS interface from the functional part of the application, our system can execute the same application program at different locations, but in different QoS domains. Thus different clients can define different staleness thresholds based on their constraints and availabilities. But at the same time, the system strives to provide strong consistency through its consistency protocols. It also tries to optimize the communication based on consistency policies used. Also at runtime, the system can adaptively change its timeliness requirements. Object caching has also been explained in systems like *Thor* [20] and *Rover* [17].

A lot of consistency related work has been done in the areas of distributed shared memory, distributed file systems and the world-wide web. Consistency protocols for the web are described in [15, 21, 28]. Weak consistency protocols based on time to live (TTL) are presented in [15]. Although these weaker consistency models provides better scalability and enhance system performance, their notions of consistency are too week to support interactive applications. Stronger notions of consistency based on invalidation and polling for the web are presented in [21]. Different distributed file systems like

XFS [9] employ caching and replication to enhance performance. Thef invalidation based consistency protocols used by them often disallow the co-existence of readers and writers and are more expensive to implement. The LC_{set} protocol described in this paper provides strong consistency but allows readers and a writer to co-exist, thus providing consistency vs. cost tradeoffs.

Also LC_{set} provides consistency across a group of objects, piggybacking information about the the object group with server's response to a client's request. It also exploits timeliness to provide better responses to user invocations and can be effective in applications where a bounded delay in the perception of the writes at the remote readers does not affect the correctness. *Coda* [18] file system addresses issues related to caching and replication in a mobile environment. Hence the consistency model allows disconnected operations to continue execution even in the presence of network partitions or failures. So in coda, it is possible to have multiple writers co-exising and hence concurrent updates that cannot be serialized. Such violations can be resolved through pre-specified policies. If such conflicts cannot be resolved, then the system intimates the user to initiate the reconcilation procedure. LC_{set} does not allow more than one writer to exist at any instant. So there will not be any conflicts in the serialization of LC_{set}. Distributed shared memory systems like *Munin* [8] *Stanford DASH* [12] provides access to shared data only after the execution of synchronization specific operations (e.g., locking) are complete. But this may not be a viable option for distributed applications because of the heavy penalty involved in executing synchronization operations in a wide-area system due to high network penalties. In comparison, the LC_{set} protocol minimizes the number of consistency messages by not transmitting the information about a new update to the client, but rather delaying it and sending it along with the client's request at a later time.

7 Conclusions

Interactive distributed applications that involve multiple users interacting with each other can be developed as shared state applications. For acceptable performance of these applications, the latency of user actions should be bound. One common approach in developing these shared state applications is to assume a distributed object abstraction for the applications shared state. A very efficient implementation of these abstractions using aggressive caching can substantially reduce the invocation times for user actions. Also, by relaxing the consistency requirements for the shared object state, a better response time for the user actions can be achieved.

In this paper we have presented the architecture of an object caching system that transparently caches objects. The framework is configurable and multiple consistency protocols are used to govern the state of the cached objects based on application requirements and resource availability. The high level specification of the consistency quality of service requirements is done through the Quality Object's Quality Description Languages. We also presented the runtime and the compile features of the system for the specification and the governance of the consistency QoS requirements.

We discuss two consistency protocols, a server based strict consistency protocol, SC_{inv} which provides a very strict serialization order for all the reads and writes in the system and the LC_{set} protocol which provides a more relaxed ordering for reads depending on the value of the timeliness threshold.

The LC_{set} protocol provides better response time for invocations, almost as small as an invocation on a local object, for interactive workloads, for a timeliness threshold value greater than 1 second. Also the number of consistency messages needed for the LC_{set} protocol is much lower than the the SC_{inv} protocol for higher timeliness thresholds.

In the future we would like to develop additional consistency protocols for the framework providing different levels of guarantees based on the timeliness and the ordering requirements. Also we would like to parameterize the interactive applications, develop synthetic workloads and simulations based on some their interesting properties and do a detailed evaluation of the protocol implementations using the workloads.

References

[1] M. Ahamad and R. Kordale. Scalable consistency protocols for distributed services. *IEEE Transcations on Parallel and Distributed Systems*, 1999.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. W. Hutto, and P. Kohli. Causal memory: Definitions, implementation and programming. *Distributed Computing*, 9:37–49, 1995.

[3] D. A. andJ. W. Barrus, J. W. Barrus, J. W. Howard, C. Rich, C. Shen, and R. C. Waters. Building multi-user interactive multimedia environments in merl. *IEEE Multimedia*, 2(4):77–82, 1995.

[4] S. Bhola and M. Ahamad. Workload modeling for highly interactive distributed applications. Technical Report GIT-CC-99-2, College of Computing, Georgia Institute of Technology, 1999.

[5] M. Blaze. *Caching in large-scale distributed file systems*. PhD thesis, Princeton, 1992.

[6] A. Bruckman. Community support for constructionist learing. In *Proc. of the 7th ACM Conference on Computer Supported Cooperative Work*, 1998.

[7] J. Calvin, A. Dickens, B. Gaines, P. Metzger, D. Miller, and D. Owen. The simnet virtual world architecture. In *Proceedings of the IEEE VRAIS*, pages 450–455, 1993.

[8] J. Carter, J. Bennett, and W. Zwaenepoel. Implementation and performance of munin. In *Proceedings of the Thirteenth Symposium on Operating Systems Principles (SOSP)*, pages 152–164, October 1991.

[9] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proc. of ACM SIGMETRICS*, 1994.

[10] J. H. H. et.al. Scale and performance in distributed file systems. *ACM Transactions on Computer Systems*, Feburary 1988.

[11] R. M. Fujimoto and R. M. Weatherly. Time management in the dod high level architecture. In *Proceedings of the Tenth Work-shop on Parallel and Distributed Simulations*, pages 60–67, 1996.

[12] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessey. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of ISCA*, 1990.

[13] S. Ghee. dvs: A distributed vr systems infrastructure. In *ACM SIGGRAPH Course Notes*, 1995.

[14] R. Gossweiler, R. J. Laferriere, M. L. Keller, and Paush. An introductory tutorial for developing multiuser virtual environments. *Presence: Teleoperators and Virtual Environments*, 3(4), 1990.

[15] J. Gwertzman and M. Seltzer. World-wide web cache consistency. In *Proc. of the 1996 USENIX Technical Conference, Jan 1996*, 1996.

[16] O. Hagsand. Interactive multiuser ves in the dive system. *IEEE Multimedia*, 1996.

[17] A. D. Joseph, A. F. de Lespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek. Rover: A toolkit for mobile information access. In *Proceedings of 15th SOSP)*, 1995.

[18] J. Kistler and M. Satyanarayanan. Disconnected operation in coda file system. In *ACM Symposium on Operating systems and Principles",)*, 1992.

[19] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 1989.

[20] B. Liskov, A. Adya, M. Castro, M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, and L. Shirra. Safe and efficient sharing of persistenct objects in thor. In *ACM SIGMOD*, 1996.

[21] C. Liu and P. Cao. Maintaining strong consistency in the world-wide web. In *Proc. of the International Conference on Distributed Computing Systems*, 1997.

[22] M. R. Macedonia, M. Z. Zyda, D. R. Pratt, P. T. Barham, and S. Zeswitz. Npsnet: A network software architecture for large-scale virtual environments. *Presence*, 3(4):265–287, 1990.

[23] M. N. Nelson, B. B. Welch, and J. K. Osterhout. Caching in sprite file sytem. *ACM Transactions on Computer Systems*, 1988.

[24] P.Cao. *A Collection of Web Proxy/Client Traces*. http://www.cs.wisc.edu/ cao/icache/proxytrace.html.

[25] P.Cao. *A Collection of Web Server Traces*. http://www.cs.wisc.edu/ cao/icache/trace.html.

[26] R. Sandberg, D. Boldberg, S. Kleiman, D. Walsh, and B.Lyon. Design and implementation of sun network filesystem. In *Proc. of the Summer Usenix Conference*, 1995.

[27] Q. Wang, M. Green, and C. Shaw. Em - an environment manager for building networked virtual environments. In *Proceedings of IEEE VRAIS*, 1995.

[28] K. Worrell. Invalidation in large scale network object caches. Master's thesis, University of Colarado, 1994.

[29] J. Zinky, D.Bakken, and R.Schantz. Architectural support for quality of service for corba objects. In *Theory and Practice of Object Systems*, pages 41–49, 1997.